Efficient Computation of Polynomial Explanations of Why-Not Questions

Nicole Bidoit, Melanie Herschel, Aikaterini Tzompanaki

Efficient Computation of Polynomial Explanations of Why-Not Questions

Nicole Bidoit
Université Paris Sud / Inria
91405 Orsay Cedex, France
nicole.bidoit@lri.fr

Melanie Herschel
Universität Stuttgart
70569 Stuttgart, Germany
melanie.herschel
@ipvs.uni-stuttgart.de

Aikaterini Tzompanaki
Université Paris Sud / Inria
91405 Orsay Cedex, France
katerina.tzompanaki@lri.fr

ABSTRACT

Answering a Why-Not question consists in explaining why a query result does not contain some expected data, called missing answers. This paper focuses on processing Why-Not questions in a query-based approach that identifies the culprit query components. Our first contribution is a general definition of a Why-Not explanation by means of a polynomial. Intuitively, the polynomial provides all possible explanations to explore in order to recover the missing answers, together with an estimation of the number of recoverable answers. Moreover, this formalism allows us to represent Why-Not explanations in a unified way for extended relational models with probabilistic or bag semantics. We further present an algorithm to efficiently compute the polynomial for a given Why-Not question. An experimental evaluation demonstrates the practicality of the solution both in terms of efficiency and explanation quality, compared to existing algorithms.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

Why-Not question, explanation, provenance

1. INTRODUCTION

The increasing load of data produced nowadays is coupled with an increasing need for complex data transformations that developers design to process these data in every-day tasks. These transformations, commonly specified declaratively, may result in unexpected outcomes. For instance, given the sample query and data of Fig. 1 on airlines and destination countries, a developer (or traveller) may wonder why Emirates does not appear in the result. Traditionally, she would repeatedly manually analyze the query to identify a possible reason, fix it, and test it to check whether the *missing answer* is now present or if other problems need to be fixed.

```
SELECT airline,
       country
FROM Airline A,
     Country C
WHERE ccode = code
AND year < 1985
```

Airline

airline	year	ccode
KLM	1919	1
Qatar	1993	1
Aegean	1987	2
Emirates	1985	3

Country

code	country
1	*Australia*
2	*France*

Figure 1: Example query and data

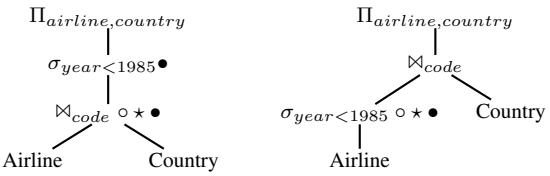


Figure 2: Reordered query trees for the query of Fig. 1 and algorithm results (Why-Not ○, NedExplain ⋆, Conseil •)

Answering such *Why-Not questions*, that is, understanding why some data are *not* part of the result, is valuable in a series of applications, such as query debugging and refinement, data verification or what-if analysis. To help developers *explain missing answers*, different algorithms have recently been proposed for relational and SQL queries and other types of queries (top-k, reverse skyline).

For relational queries, Why-Not questions can be answered for example based on the data (instance-based explanations), the query (query-based explanations), or both (hybrid explanations). We focus on solutions producing query-based explanations, as these are generally more efficient while providing sufficient information for query analysis and debugging. Essentially, a query-based explanation is a set of query conditions that are responsible for pruning out data relevant to the missing answers. Existing methods producing query-based explanations are not satisfactory as they are designed over query trees, making the explanations depending on the topology of a given tree. Consequently, they return different explanations for the same SQL query and may miss explanations.

EXAMPLE 1.1. *Consider again the SQL query and data of Fig. 1 and assume that a developer wants an explanation for the absence of Emirates from the query result. Fig. 2 shows two possible query trees. It also shows the tree operators that Why-Not [7] (○) and NedExplain [3] (⋆) return as query-based explanations as well as the tree operators returned as part of hybrid explanations by Conseil [13, 14] (•). Each algorithm returns a different result for each of the two query trees, and in most cases, it is only a partial result as the true explanation of the missing answer is that* both *the selection is too strict for the tuple* $(Emirates, 1985, 3)$ *from table* $Airline$ *and this tuple does not find join partners in table* $Country$*.*

The above example clearly shows the shortcomings of existing algorithms. Indeed, the developer first has to understand and reason at the level of query trees instead of reasoning at the level of the declarative SQL query she is familiar with. Second, she always has to wonder whether the explanation is complete, or if there are other explanations that she could consider instead. In this paper, we make the following contributions:

Extended formalization of Why-Not explanation polynomial. We have recently introduced polynomials as Why-Not explanations in the context of the relational model under set semantics [2]. A polynomial provides a complete explanation and is independent of a specific query tree representation, solving the problems illustrated by Ex. 1.1. We now extend Why-Not explanations to the relational model under bag and probabilistic semantics. This confirms the robustness of the chosen polynomial representation, making it a good fit for a unified framework for representing Why-Not explanations. In parallel, we considerably simplify our initial framework by eliminating the formerly used notion of query tableaux.

Efficient *Ted++* algorithm. We show that a naive algorithm computing Why-Not explanations, as presented in [2], is impractical. We thus propose a novel algorithm, *Ted++*, capable of efficiently computing the Why-Not explanation polynomial, based on techniques like schema and data partitioning (allowing for a distributed computation) and advantageous replacement of expensive database evaluations by mathematical calculations.

Experimental validation. We validate our solutions both in terms of efficiency and effectiveness. Our experiments include a comparative evaluation to existing algorithms computing query-based explanations for SQL queries (or sub-languages thereof) as well as a thorough study of *Ted++* performance w.r.t. different parameters. Note that such an evaluation was missing from [2].

The remainder of this paper is structured as follows. Sec. 2 covers related work. Sec. 3 defines in detail our problem setting and the Why-Not explanation polynomials. Next, we discuss the *Ted++* algorithm in Sec 4. Finally, we present our experimental setup and evaluation in Sec. 5 and conclude in Sec. 6.

2. RELATED WORK

The work presented in this paper falls in the category of data provenance research and specifically on explaining missing answers from query results. Due to the lack of space, we focus here on this sub-problem, thus on algorithms answering Why-Not questions, summarized in Tab. 1. This table classifies the algorithms according to the type of explanation they generate and reports the class of query and Why-Not question (simple or complex)[1] they support.

Query-based and hybrid explanations. Why-Not [7] takes as input a simple Why-Not question and a selection, projection, join, and union (SPJU) query and returns the erroneous query operators as query-based explanations. Similarly, NedExplain [3] considers selection, projection, join, and aggregation and unions thereof (SPJUA queries) and simple Why-Not questions as well. The common drawback of the two algorithms is that their design is dependent on a specific query tree representation, thus the explanations proposed are tied to this tree. Moreover, the generated explanations are incomplete. To address this problem, Ted [2] proposes explanations in the form of a polynomial. The shortcomings of this work have already been presented in Sec. 1.

Conseil [13, 14] produces hybrid explanations that include an instance-based and a query-based component. The latter consists in a set of picky query operators. However, as Conseil considers both the data to be possibly incomplete and the query to be possibly faulty, the set of picky query operators associated to a hybrid explanation depends on the set of source edits of the same hybrid explanation.

[1]A *simple* Why-Not question involves conditions that impact one relation only, otherwise it is *complex* (see Sec. 3).

Table 1: Algorithms for answering Why-Not questions

Algorithm	Why-Not question	Explanation format	Query
Query-based explanations			
Why-Not [7]	simple	query operators	SPJU
NedExplain [3]	simple	query operators	SPJUA
Ted [2]	complex	polynomial	conj. queries with inequalities
Hybrid explanations			
Conseil [13, 14]	simple	source table edits + query operators	SPJAN
Instance-based explanations			
MA [16]	simple	source table edits	SPJ
Artemis [15]	complex	source table edits	SPJUA
Meliou *et. al.* [19]	simple	causes (tuples) and responsibility	conjunctive queries
Calvanese *et. al.* [5]	simple	additions to ABox	instance & conj. queries over DL-Lite ontology
Ontology-based explanations			
Cate *et. al.* [6]	simple	tuples concepts	conj. queries with comparisons
Refinement-based explanations			
ConQueR [20]	complex	refined query	SPJA
TALOS [21]			SPJ
FlexIQ [17]	simple	refined query & Why-Not question	SPJ
Zhang *et. al.* [12]	simple	refined query	Top-k query
Islam *et. al.* [18]	simple	refined query & Why-Not question	Reverse skyline query
WQRTQ [9]	simple	refined query & Why-Not question	Reverse Top-k query
Chen *et. al.* [8]	simple	refined query	Spatial keyword Top-k query

Instance-based explanations. Missing-Answers (MA) [16] and Artemis [15] compute explanations in the form of source table edits for SPJ queries and SPJUA queries respectively. Meliou *et. al.* [19] study the concepts of causality and responsibility of instance-based explanations for data present or missing in a conjunctive query result. Calvanese *et. al.* [5] compute instance-based explanations on data represented by a DL-Lite ontology.

Ontology-based explanations. Cate *et. al.* [6] have recently introduced this type of explanation that is based on external or data-workload generated ontologies. However, they are completely independent of the query to be analyzed.

Refinement-based explanations. Another approach to answering Why-Not questions is by directly proposing queries that include in their result the missing answers. Several algorithms follow this direction for different types of queries, like relational, Top-K, reverse sklyline queries, etc, as listed in Tab. 1. Although these approaches are generally very interesting, they do not focus on pinpointing the erroneous parts of the query. Indeed, the refined queries may contain changes that are not necessarily tied to an erroneous part of the query. Moreover, the changes are based on the database values and do not take into account any semantics or domain knowledge that could render a refinement meaningful for the user.

3. WHY-NOT EXPLANATION POLYNOMIAL

This section introduces a polynomial formalization of query-based Why-Not explanations. We assume the reader familiar with the relational model [1], and we only briefly revisit some relevant notions in Sec. 3.1 while we formalize Why-Not questions. In Sec. 3.2, we define the explanation of a Why-Not question as a polynomial. In Sec. 3.3 we provide a unified general framework for Why-Not explanations in the context of set, bag, and probabilistic semantics databases.

R

A	B	R.Id
1	3	Id_1
2	4	Id_2
4	5	Id_3
8	9	Id_4

S

D	E	S.Id
4	8	Id_5
5	3	Id_6
3	9	Id_7

T

B	C	D	T.Id
3	4	5	Id_8
3	8	1	Id_9
5	3	3	Id_{10}
5	9	4	Id_{11}

(a) Sample Instance $\mathcal{I}$

$\mathcal{S} = \{R, S, T\}$

$\Gamma = \{R.B, S.D, T.C\}$

$C = \{c_1, c_2, c_3, c_4, c_5\}$

	C
c_1	$R.A > 3$
c_2	$R.B = T.B$
c_3	$T.C \geq 8$
c_4	$T.D = S.D$
c_5	$S.E \geq 3$

$\{R.B<S.D, T.C \leq 9\}$

(c) Why-Not question WN

(b) query $Q = (\mathcal{S}, \Gamma, C)$ and naming of conditions

Figure 3: Running example

3.1 Preliminaries

For the moment, we limit our discussion to relational databases under set semantics. A database schema $\mathcal{S}$ is a set of relation schemas. A relation schema R is a set of attributes. We assume each attribute of R qualified, i.e., of the form $R.A$ and for the sake of simplicity we assume a unique domain Dom. $\mathcal{I}$ denotes a database instance over $\mathcal{S}$ and $\mathcal{I}_{|R}$ denotes the instance of a relation $R \in \mathcal{S}$. We assume that each database relation R has a special attribute $R.Id$, which is used as identifier for the tuples in $\mathcal{I}_{|R}$. For any object O (relational or database schema, condition etc), $\mathcal{A}(O)$ denotes the set of attributes occurring in O. Finally, a condition c over $\mathcal{S}$ is defined as an expression of the form $R.A\ \theta\ a$ where $a \in Dom$ or of the form $R.A\ \theta\ S.B$, where $R.A, S.B \in \mathcal{A}(\mathcal{S})$, and $\theta \in \{=, \neq, <, \leq\}$. A condition over two relations is *complex*, otherwise it is *simple*. In this article, we consider conjunctive queries with inequalities. Note that in our approach, the database schema $\mathcal{S}$ denotes the query input schema. In an SQL-like approach, each time we need an instance of a relation, we refer to it by a different name. In this way, we are able to correctly define Why-Not questions in case of self-joins as well.

DEFINITION 3.1 (QUERY Q). *A query Q is specified by the triple $(\mathcal{S}, \Gamma, C)$, where $\mathcal{S}$ is a database schema, $\Gamma \subseteq \mathcal{A}(\mathcal{S})$ is the projection attribute set, and C is a set of conditions over $\mathcal{A}(\mathcal{S})$. The semantics of Q are given by the relational algebra expression $\pi_\Gamma[\sigma_{\bigwedge_{c \in C} c}[\times_{R \in \mathcal{S}}[R]]]$.*

The result of Q over $\mathcal{I}$ is denoted by $Q[\mathcal{I}]$. Note here that we are not concerned about the evaluation/optimization of Q and that any equivalent rewriting of the algebraic expression given in Def. 3.1 is a candidate for evaluating Q.

EXAMPLE 3.1. *Fig. 3 describes our running example. Fig. 3(a) displays an instance $\mathcal{I}$ over $\mathcal{S}=\{R, S, T\}$. Fig. 3(b) displays a query Q over $\mathcal{S}$, whose conditions have been named for convenience. $R.B=T.B$ and $T.D=S.D$ are complex whereas the others are simple conditions. Moreover, $Q[\mathcal{I}]=\{(R.B{:}5, S.D{:}4, T.C{:}9)\}$.*

In our framework, a Why-Not question specifies missing tuples from the result of a query Q through a conjunctive set of conditions. A Why-Not question is related to the result of Q and so its conditions are restricted to the attributes of the output schema of Q.

DEFINITION 3.2 (WHY-NOT QUESTION). *A Why-Not question WN w.r.t. Q is defined as a set of conditions over Γ.*

The notion of complex and simple conditions is extended to complex and simple Why-Not questions in a straightforward manner. Due to lack of space, we do not provide here more real-world examples of Why-Not questions and refer the reader to scenarios in [4].

As we said, a Why-Not question WN summarizes a set of (missing) tuples that the user expected to find in the query result. To be able to obtain these missing tuples as query results, data from the input relation instances that satisfy WN need to be combined by the query. The candidate data combinations are what we call *compatible* tuples and can be computed using WN as in Def. 3.3.

DEFINITION 3.3 (COMPATIBLE TUPLES). *Consider the query $Q_{WN}=(\mathcal{S}, \mathcal{A}(\mathcal{S}), WN)$, where $\mathcal{S}$ is also the input schema of Q. The set CT of compatible tuples is the result of the query Q_{WN} over $\mathcal{I}$.*

We further introduce the notion of a *well founded* Why-Not question. Intuitively, a Why-Not question can be answered under a query-based approach, only if some data in $\mathcal{I}$ match the Why-Not question (otherwise instance-based explanations should be sought for). Moreover, a Why-Not question is meaningful if it tracks data not already returned by the query.

DEFINITION 3.4 (WELL FOUNDED WHY-NOT QUESTION). *A Why-Not question WN is said to be well founded if $CT \neq \emptyset$ and $\pi_\Gamma[CT] \cap Q[\mathcal{I}] = \emptyset$.*

EXAMPLE 3.2. *Continuing Ex. 3.1, we may wonder why there is not a tuple for which $R.B<S.D$ and $T.C \leq 9$. According to Def. 3.2, this Why-Not question can be seen as the conjunction of the conditions $R.B<S.D \wedge T.C \leq 9$ (Fig. 3(c)). Since $R.B<S.D$ is a complex condition, WN is a complex Why-Not question. The compatible tuples set CT is the result of the query $Q_{WN}=\sigma_{R.B<S.D \wedge T.C \leq 9}[R \times S \times T]$, which contains 12 tuples. For example, one compatible tuple is $\tau_1=(R.Id{:}1, R.A{:}1, R.B{:}3, S.Id{:}5, S.D{:}4, S.E{:}8, T.Id{:}8, T.B{:}3, T.C{:}4, T.D{:}5)$.*

Each tuple in CT could have led to a missing tuple, if it was not eliminated by some of the query conditions. Thus, explaining WN amounts to identifying these blocking query conditions.

3.2 Why-Not Explanation

To build the query-based explanation of WN, we start by specifying what explains that a compatible tuple τ did not lead to an answer. Intuitively, the explanation consists of the query conditions pruning out τ.

DEFINITION 3.5 (EXPLANATION FOR τ). *Let $\tau \in CT$ be a compatible tuple w.r.t. WN, given Q. Then, the explanation for τ is the set of conditions $\mathcal{E}_\tau=\{c | c \in C \text{ and } \tau \not\models c\}$.*

EXAMPLE 3.3. *Consider the compatible tuple τ_1 in Ex. 3.2. The conditions of Q (see Ex. 3.1), not satisfied by τ_1 are: c_1, c_3, and c_4. So, the explanation for τ_1 is $\mathcal{E}_{\tau_1}=\{c_1, c_3, c_4\}$.*

Having defined the explanation w.r.t. one compatible tuple, the explanation for WN is obtained by simply summing up the explanations for all the compatible tuples in CT, leading to the expression $\sum_{\tau \in CT} \prod_{c \in \mathcal{E}_\tau} c$. We justify modelling the explanation of τ with a product (meaning conjunction) of conditions by the fact that in order for τ to 'survive' the query conditions and give rise to a missing tuple, every single condition in the explanation must be 'repaired'. The sum (meaning disjunction) of the products for each $\tau \in CT$ means that if any explanation is 'correctly repaired', the associated τ will produce a missing tuple.

Of course, several compatible tuples can share the same explanation. Thus, the final Why-Not explanation is a polynomial having as variables the query conditions and as integer coefficients the number of compatible tuples sharing an explanation.

DEFINITION 3.6 (WHY-NOT EXPLANATION). *With the same assumption as before, the Why-Not explanation for WN is defined as the polynomial*

$$\text{PEX} = \sum_{\mathcal{E} \in E} coef_{\mathcal{E}} \prod_{c \in \mathcal{E}} c$$

where $E = 2^C$ *and* $coef_{\mathcal{E}} \in \{0, \ldots, |CT|\}$.

Intuitively, E contains all potential explanations, and each of these explanations prunes from zero to at most $|CT|$ compatible tuples. Moreover, an important property of *PEX* is the fact that $coef_{\mathcal{E}} = |\ \{\tau \in CT | \mathcal{E}$ is the explanation for $\tau\}\ |$, meaning that $coef_{\mathcal{E}}$ equals the number of compatible tuples with the same explanation.

Each term of the polynomial provides an alternative explanation to be explored by the user who wishes to recover some missing tuples. Additionally, the polynomial offers, through its coefficients, some useful hints to users interested in the *number* of recoverable tuples. More precisely, by choosing an explanation $\mathcal{E}$ to repair, we obtain an upper bound for the number of compatible tuples that can be recovered. The upper bound is the sum of the coefficients of all the explanations that are sub-sets of (the set of conditions of) $\mathcal{E}$. Consequently, the coefficients could be used to answer Why-Not questions of the form *Why-Not $x missing tuples?*.

EXAMPLE 3.4. *In Ex. 3.3 we found the explanation* $\{c_1, c_3, c_4\}$, *which is translated to the polynomial term* $c_1 * c_3 * c_4$. *Taking into consideration all the 12 compatible tuples of our example, we obtain the following* PEX *polynomial:* $2*c_1*c_4 + 2*c_1*c_3*c_4 + 4*c_1*c_2*c_4 + 2*c_1*c_2*c_3 + 2*c_1*c_2*c_3*c_4$. *In the polynomial, each addend, composed by a coefficient and an explanation, captures a way to obtain missing tuples. For instance, the explanation* $c_1 * c_2 * c_4$ *indicates that we may recover some missing answers if* c_1 *and* c_2 *and* c_4 *are changed. Then, the sum of its coefficient 4 and the coefficient 2 of the explanation* $c_1 * c_4$ *(*$\{c_1, c_4\} \subseteq \{c_1, c_2, c_4\}$*) indicates that we can recover from 0 to 6 tuples.*

As the visualization of the polynomial per se may be cumbersome and thus not easy for a user to manipulate, some post-processing steps could be applied. Depending on the application or needs, only a subset of the explanations could be returned like for instance minimum explanations (i.e., for which no sub-explanations exist), or explanations giving the opportunity to recover a specific number of tuples, or have specific condition types etc.

3.3 Extension: Bag & Probabilistic Semantics

So far, we have considered databases under *set* semantics only. In this section, we discuss how the definition of Why-Not explanation (Def. 3.6) extends to settings with conjunctive queries over bag and probabilistic semantics.

K-relations, as introduced in [10], capture in a unified manner relations under set, bag, and probabilistic semantics. Briefly, tuples in a K-relation are annotated with elements in K. In our case, we consider that K is a set of unique tuple identifiers, similar to our special attribute $R.Id$ in Sec. 3.1.

In what follows, we use the notion of *how*-provenance of tuples in the result of a query Q. The how-provenance of $t \in Q(\mathcal{I})$ is modelled as the polynomial obtained by the positive algebra on K-relations, proposed in [10]. Briefly, each t is annotated with a polynomial where variables are tuple identifiers and coefficients are natural numbers. Roughly, if t results from a selection operator on t_1 annotated with Id_1, then t is also annotated with Id_1. If t is the result of the join of t_1 and t_2, then t is annotated with $Id_1 Id_2$.

We compute the *generalized* Why-Not explanation polynomial PEX_{gen} as follows. Firstly, we compute the how-provenance for compatible tuples in CT by evaluation of the query Q_{WN} (Def. 3.3) w.r.t. the algebra in [10]. Recall that Q_{WN} contains only selection and join operators. Thus, each compatible tuple τ in CT is annotated with its how-provenance polynomial, denoted by η_τ.

Then, we associate the expressions of *how* and *why-not* provenance. In order to do this, for each compatible tuple τ in CT, we combine its how-provenance polynomial η_τ with its explanation $\mathcal{E}_\tau$ (Def. 3.5). So, each τ is associated with the expression $\eta_\tau \mathcal{E}_\tau$.

Finally, we sum the combined expressions for all compatible tuples, which leads to the expression $\sum_{\tau \in CT} \eta_\tau \mathcal{E}_\tau$.

We now briefly comment on how PEX_{gen} is instantiated to deal either with the set, bag, or probabilistic semantics. Indeed, the 'specialization' of PEX_{gen} relies on the interpretation of the elements in K, that is on a function $Eval$ from K to some set L. For the set semantics, each tuple in a relation occurs only once. This results in choosing L to be the singleton $\{1\}$ and mapping each tuple identifier to 1. It is then quite obvious to note, for the set semantics, that $PEX_{gen} = PEX$ (Def. 3.6). In the same spirit, for bag semantics, L is chosen as the set of natural numbers $\mathbb{N}$ and each tuple identifier is mapped to its number of occurences. Finally, for probabilistic databases, L is chosen as the interval $[0, 1]$ and each tuple identifier is mapped to its occurrence probability.

Thus, the generalized definition of Why-Not explanation is parametrized by the mapping $Eval$ of the annotations (elements in K) in the set L.

DEFINITION 3.7. *(Generalized Why-Not explanation polynomial) Given a query* Q *over a database schema* $\mathcal{S}$ *of* K*-relations, the generalized Why-Not explanation polynomial for WN is*

$$\text{PEX}_{gen} = \sum_{\mathcal{E} \in E} \Big(\sum_{\tau \in CT s.t.\ \mathcal{E}_\tau = \mathcal{E}} Eval(\eta_\tau) \Big) \mathcal{E}$$

where $E = 2^C$, η_τ *is the how-provenance of* τ, *and* $Eval: K \rightarrow L$ *maps the elements of* K *to values in* L.

4. TED++ ALGORITHM

The naive Ted algorithm [2] implements the definitions of [2] for Why-Not explanations in a straightforward manner. Essentially, Ted first enumerates the set of compatible tuples. Then, it computes the explanation for each compatible tuple, leading to the computation of the final Why-Not explanation. However, both of these steps make Ted computationally prohibitive. Not only is the computation of the set of compatible tuples time and space consuming as it often requires cross product executions, but the same holds for the iteration over this (potentially very large) set. Ted's time complexity is $O(n^{|\mathcal{S}|})$, $n = max(\{|\ \mathcal{I}_R\ |\}), R \in \mathcal{S}$. As experiments in Sec. 5 confirm, this complexity renders Ted impractical.

To overcome Ted's poor performance, we propose *Ted++*. The main feature of *Ted++* is to completely avoid enumerating and iterating over the set CT, thus it significantly reduces both space and time consumption. Instead, *Ted++* opts for (i) iterating over the space of possible explanations, which is expected to be much smaller, (ii) computing *partial* sets of *passing* compatible tuples, and (iii) computing the *number* of *eliminated* compatible tuples for each explanation. Intuitively, passing tuples w.r.t. an explanation are tuples satisfying the conditions of the explanation. Finally, we compute the polynomial based on mathematical calculations.

Alg. 1 provides an outline of *Ted++*. The input includes the query $Q = (\mathcal{S}, \Gamma, C)$, the Why-Not question WN and the input instance $\mathcal{I}$. Firstly in Alg. 1, line 1, all potential explanations (combinations of the conditions in C) are enumerated ($E = 2^C$). The remaining steps, discussed in the next subsections, aim at computing the coefficient of each explanation. To illustrate the concepts

Algorithm 1: *Ted++*

Input: $Q{=}(\mathcal{S}, \Gamma, C), \mathcal{I}, WN$
Output: *PEX*
1 $E \leftarrow$ powerset(C);
2 $\mathcal{P} \leftarrow$validPartitioning($\mathcal{S}, WN$); * Def. 4.1 *
3 **for** $Part$ *in* $\mathcal{P}$ **do**
4 $\quad CT_{|Part} \leftarrow (Part, \mathcal{A}(Part), WN_{|Part})[\mathcal{I}_{|Part}]$;
5 coefficientEstimation(E, $Partition$);
6 *PEX* $\leftarrow$post-processing(); * Eq. (F) *
7 **return** PEX;

introduced in the detailed discussions, we will rely on our running example, for which Fig. 4 shows all relevant intermediate results. It should be read bottom-up. For convenience, in our examples, we use subscript i instead of c_i.

The subsequent discussion on *Ted++* can be considered as a proof sketch of the following theorem.

THEOREM 4.1. *Given a query Q, a Why-Not question WN and an input instance $\mathcal{I}$,* Ted++ *computes exactly* PEX.

4.1 Partial Compatible Tuples Computation

Using the conditions in WN, *Ted++* partitions the schema $\mathcal{S}$ (Alg. 1, line 2) into components of relations connected by the conditions in WN (Def. 4.1).

DEFINITION 4.1. *(Valid Partitioning of $\mathcal{S}$). Given WN, the partitioning of a database schema $\mathcal{S}$ into k partitions, denoted $\mathcal{P} = \{Part_1, \ldots, Part_k\}$, is* valid *if each $Part_i$, $i{\in}\{1, \ldots, k\}$ is minimal w.r.t. the following property:*
if $R{\in}Part_i$ and $R'{\in}\mathcal{S}$ s.t. $\exists c{\in}WN$ with $\mathcal{A}(c){\cap}\mathcal{A}(R'){\neq}\emptyset$ and $\mathcal{A}(c){\cap}\mathcal{A}(R){\neq}\emptyset$ then $R'{\in}Part_i$.

The partitioning of $\mathcal{S}$ allows for handling compatible tuples more efficiently, by 'cutting' them in distinct meaningful 'chunks'. We refer to chunks of compatible tuples as *partial* compatible tuples and group them in sets depending on the partition they belong to.

The set $CT_{|Part}$, where $Part{\in}\mathcal{P}$ is obtained by evaluating the query $Q_{Part}{=}(Part, \mathcal{A}(Part), WN_{|Part})$ over $\mathcal{I}_{|Part}$ (Alg. 1, line 4). $WN_{|Part}$ and $\mathcal{I}_{|Part}$ denote the restriction of WN and $\mathcal{I}$ over the relations in $Part$, respectively.

EXAMPLE 4.1. *The valid partitioning of $\mathcal{S}$ is $Part_1{=}\{R, S\}$ (because of the condition $R.B{<}S.D$) and $Part_2{=}\{T\}$. The sets of partial compatible tuples $CT_{|Part_1}$ and $CT_{|Part_2}$ are given in the bottom line of Fig. 4.*

It is easy to prove that the valid partitioning of $\mathcal{S}$ is unique and that the set CT can be computed from the sets $CT_{|Part}$.

LEMMA 4.1. *Let $\mathcal{P}$ be the valid partitioning of $\mathcal{S}$. Then,* $CT{=}\times_{Part{\in}\mathcal{P}}\,[CT_{|Part}]$.

Lemma. 4.1 makes it clear how to compute CT from partial compatible tuples. Our algorithm is designed in a way that avoids computing CT and relies on the computation of $CT_{|Part}$ only.

Next, we compute the number of compatible tuples pruned by each potential explanation, using the partial compatible tuple sets. In this way we calculate the coefficient of the terms in the polynomial. Since from this point on we are only handling compatible tuples, we omit the word 'compatible' to lighten the discussion.

Algorithm 2: coefficientEstimation

Input: E explanations space, $\mathcal{P}$ valid partitioning of $\mathcal{S}$
1 **for** $\mathcal{E}{\in}E$ **access in ascending size order** **do**
2 $\quad$ Compute $part_{\mathcal{E}}$;
3 $\quad$ **if** $|\ \mathcal{E}\ | = 1$ **then**
4 $\quad\quad$ materialize $V_{\mathcal{E}}$;
5 $\quad\quad$ $\beta_{\mathcal{E}} \leftarrow$ Eq. (B);
6 $\quad$ **else**
7 $\quad\quad$ **if** $\alpha_{subcombination\ of\ \mathcal{E}} \neq 0$ **then**
8 $\quad\quad\quad$ $\{\mathcal{E}1,\mathcal{E}2\} \leftarrow$ subCombinationsOf($\mathcal{E}$);
9 $\quad\quad\quad$ $\Gamma_{12} \leftarrow \Gamma_1 \cap \Gamma_2$; *$\Gamma_i$ is the output schema of $V_{\mathcal{E}i}$*
10 $\quad\quad\quad$ **if** $\Gamma_{12} \neq \emptyset$ **then**
11 $\quad\quad\quad\quad$ $V_{\mathcal{E}} \leftarrow V_{\mathcal{E}1} \bowtie_{\Gamma_{12}} V_{\mathcal{E}2}$;
12 $\quad\quad\quad\quad$ materialize $V_{\mathcal{E}}$;
13 $\quad\quad\quad$ **else**
14 $\quad\quad\quad\quad$ $|\ V_{\mathcal{E}}\ | \leftarrow |\ V_{\mathcal{E}1}\ | * |\ V_{\mathcal{E}2}\ |$;
15 $\quad\quad$ **else**
16 $\quad\quad\quad$ $|\ V_{\mathcal{E}}\ | \leftarrow |\ V_{\mathcal{E}1}\ | * |\ V_{\mathcal{E}2}\ |$;
17 $\quad\quad$ $\beta_{\mathcal{E}} \leftarrow \prod_{Part{\in}part_{\mathcal{E}}} |CT_{|Part}| - |\ (\bigcup_{i=1}^{n} V_{ci})^{ext}\ |$; * Eq. (E) *
18 $\quad$ $\alpha_{\mathcal{E}} \leftarrow$ Eq. (A);

4.2 Polynomial Coefficient Estimation

Each set $\mathcal{E}$ in the powerset E is in fact a potential explanation that is further processed. This process is meant to associate with $\mathcal{E}$ (i) the set of partitions $part_{\mathcal{E}}$ on which $\mathcal{E}$ is defined, (ii) the view definition $V_{\mathcal{E}}$ meant to store the passing partial tuples w.r.t. $\mathcal{E}$[2], and (iii) the number $\alpha_{\mathcal{E}}$ of tuples eliminated by $\mathcal{E}$.

Alg. 2 describes how we process E in ascending order of explanation size. This enables us to reuse results obtained for sub-explanations and in combination with mathematics, avoid cross product computations.

We first determine the set of partitions for an explanation $\mathcal{E}$ as $part_{\mathcal{E}}{=}{\cup_{c\in\mathcal{E}}}\,\{Part_c\}$, where $Part_c$ contains at least one relation over which c is specified.

EXAMPLE 4.2. *Consider $\mathcal{E}_1{=}\{c_1\}$ and $\mathcal{E}_2{=}\{c_2\}$. From Fig. 3(b) and the partitions in Fig. 4, we can see that c_1 impacts only $Part_1$, whereas c_2 spans over $Part_1$ and $Part_2$. Hence, $part_{\mathcal{E}_1}{=}\{Part_1\}$ and $part_{\mathcal{E}_2}{=}\{Part_1, Part_2\}$. Then, $\mathcal{E}{=}\{c_1, c_2\}$ is impacted by the union of $part_{\mathcal{E}_1}$ and $part_{\mathcal{E}_2}$, thus $part_{\mathcal{E}}{=}\{Part_1, Part_2\}$.*

We use Eq. (A) to calculate the number $\alpha_{\mathcal{E}}$ of eliminated tuples, using the number $\beta_{\mathcal{E}}$ of eliminated *partial* tuples and the cardinality of the partitions not in $part_{\mathcal{E}}$. Intuitively, this formula extends the partial tuples to "full" tuples over CT's schema.

$$\alpha_{\mathcal{E}} = \beta_{\mathcal{E}} * \prod_{Part{\in}\overline{part_{\mathcal{E}}}} |CT_{|Part}|, \qquad \text{(A)}$$

where $\overline{part_{\mathcal{E}}}{=}\mathcal{P} \setminus part_{\mathcal{E}}$. Note that when $\overline{part_{\mathcal{E}}}$ is empty, we abusively consider that $\prod_{\emptyset} = 1$.

The presentation now focuses on calculating $\beta_{\mathcal{E}}$. Two cases arise depending on the size of $\mathcal{E}$.

Atomic explanations. We start with atomic explanations $\mathcal{E}$ containing only one condition c (Algorithm 2 lines 3-5). We firstly compute the set of *passing* partial tuples w.r.t. c, i.e., the tuples that satisfy c, which we store in the view V_c:

$$V_c = \begin{cases} \pi_{\{R\_id|R{\in}Part\}}(\sigma_c[CT_{|Part}]) \text{ if } part_{\mathcal{E}}{=}\{Part\} \\ \pi_{\{R\_id|R{\in}Part_1{\cup}Part_2\}}([CT_{|Part_1}] \bowtie_c [CT_{|Part_2}]) \\ \qquad\qquad \text{if } part_{\mathcal{E}}{=}\{Part_1, Part_2\} \end{cases}$$

[2]We choose to store passing rather than eliminated tuples as they are usually less numerous. In an optimized version this decision could be made dynamically based on view cardinality estimation.

Then, the number of eliminated partial tuples by $\mathcal{E}$ is

$$\beta_{\mathcal{E}} = \prod_{Part \in part_{\mathcal{E}}} |CT_{|Part}| - |V_c| \quad \text{(B)}$$

EXAMPLE 4.3. *For c_2, we have $part_{c_2}=\{Part_1, Part_2\}$, so $V_{c2}=\pi_{R\_Id,S\_Id,T\_Id}([CT_{|Part_1}]\bowtie_{R.B=T.B}[CT_{|Part_2}])$. This results in $|V_{c_2}|=4$, and by Eq. (B) we obtain $\beta_{c2}=|CT_{|Part1}| * |CT_{|Part2}|-|V_{c2}|=3 * 4-4=8$. Since all partitions of $\mathcal{P}$ are in $part_{c2}$, applying Equ. (A) results in $\alpha_{c2}=\beta_{c2}=8$. For c_3, $\beta_{c3}=|CT_{|Part_2}|-V_{c3}=4-2=2$, so $\alpha_{c3}=3*2=6$. Fig. 4 (second level) displays the process for all atomic explanations.*

Non atomic explanations. Now, assume that $\mathcal{E}=\{c_1,\ldots,c_n\}$, $n > 1$ (Alg. 2, lines 6-16). For the moment, we assume that the conditions in $\mathcal{E}$ share the same schema, so the intersection and union of V_{ci} for $i = 1,\ldots,n$ are well-defined. Firstly, we compute the view $V_{\mathcal{E}}$ of passing partial tuples w.r.t. $\mathcal{E}$ as $V_{\mathcal{E}}=V_{c1}\cap\ldots\cap V_{cn}$. To compute the number of partial tuples pruned out by $\mathcal{E}$, we need to find the number of partial tuples pruned out by c_1 and ... and c_n, i.e., $\beta_{\mathcal{E}}= \mid \overline{V_{c1}} \cap \cdots \cap \overline{V_{cn}} \mid$. By the well-known *DeMorgan law* [22], we have $\beta_{\mathcal{E}}=|\overline{V_{c1}} \cup \cdots \cup \overline{V_{cn}}|$, which spares us from computing the complements of V_{ci}.

To compute the cardinality of the union of the V_{ci}, we rely on the *Principle of Inclusion and Exclusion for counting* [11]:

$$|\bigcup_{i=1}^{n} V_{c_i}| = \sum_{\emptyset\neq J\subseteq[n]} (-1)^{|J|+1} \mid \bigcap_{j\in J} V_{cj} \mid$$

We further rewrite the previous formula to reuse results obtained for sub-combinations of $\mathcal{E}$, obtaining Eq. (C).

$$\begin{aligned} |\bigcup_{i=1}^{n} V_{ci}| = |\bigcup_{i=1}^{n-1} V_{ci}| + |V_{cn}| \\ + \sum_{\emptyset\neq J\subseteq[n-1]} (-1)^{|J|} \mid \bigcap_{j\in J} V_{cj}\cap V_{cn} \mid \end{aligned} \quad \text{(C)}$$

At this point, we can compute $\beta_{\mathcal{E}}$. However, so far we assumed that the conditions in $\mathcal{E}$ have the same schema. In the general case, this does not hold and we have to "extend" the schema of a view V_c to the one of $V_{\mathcal{E}}$, in order to ensure set operations to be well-defined. The cardinality of an extended V_c^{ext} is given by Eq. (D).

$$|V_c^{ext}| = \prod_{Part\in part_{\mathcal{E}}\setminus part_c} |CT_{Part}| * |V_c| \quad \text{(D)}$$

Based on Eq. (D) we obtain Eq. (E) that generalizes Eq. (C).

$$\begin{aligned} |(\bigcup_{i=1}^{n} V_{ci})^{ext}| = |(\bigcup_{i=1}^{n-1} V_{ci})^{ext}| + |V_{cn}^{ext}| \\ + \sum_{\emptyset\neq J\subseteq[n-1]} (-1)^{|J|} \mid ((\bowtie V_{cj})_{j\in J} \bowtie V_{cn})^{ext} \mid \end{aligned} \quad \text{(E)}$$

In Eq. (E) we have replaced the intersection with natural join. The cardinalities of the views $V_{\mathcal{E}'} = (\bowtie V_{cj})_{j\in J}\bowtie V_{cn}$ associated with $\mathcal{E}'$ for $\mid J \mid < n-1$, have already been computed by previous steps and have only to be extended to the schema of $V_{\mathcal{E}}$. When $|J|=n-1$, then $V_{\mathcal{E}'}=V_{\mathcal{E}}$. A detailed discussion on how and when we materialize the view $V_{\mathcal{E}}$ is given shortly after.

Now, we trivially compute the number $\beta_{\mathcal{E}}$ of eliminated partial tuples as the complement of $\mid (\bigcup_{i=1}^{n} V_{ci})^{ext} \mid$ (Alg. 2, line 17). The number of 'full' eliminated tuples is then calculated by Eq. (A).

EXAMPLE 4.4. *To illustrate the concepts introduced above, please follow on Fig. 4 below discussion.*

For the explanation c_2c_3, Eq. (E) gives: $|(V_2\cup V_3)^{ext}|=|V_2^{ext}|+|V_3^{ext}|-|(V_2\bowtie V_3)^{ext}|$. The schema of $part_{23}=\{Part_1, Part_2\}$ is $\Gamma_{23}=\{R\_Id, S\_Id, T\_Id\}$. The view V_2 has already a matching schema, thus $|V_2^{ext}|=|V_2|=4$. For V_3, $\Gamma_3=\{T\_Id\}$, we thus apply Eq. (D) and obtain $|V_3^{ext}|=|V_{Part_1}|*|V_3|=3*2=6$. Still, $|V_{23}|=|(V_2 \bowtie V_3)^{ext}|$ remains to be calculated. Intuitively, because V_2 and V_3 target schemas share attribute T_Id, $V_{23}=V_2\bowtie_{T\_Id}V_3$. The view V_{23} is materialized and contains 2 tuples (as shown in Fig. 4). So, finally, from Eq. (E) we obtain $|(V_2 \cup V_3)^{ext}|=4+6-2=8$. Since $|Part_1| * |Part_2| = 12$ then $\beta_{23}=12-8=4$, and by Eq. (A) $\alpha_{23}=4$.

We now focus on the explanation c_3c_5. The schemas of V_3 and V_5 are disjoint and intuitively $V_{35}=V_3\bowtie V_5$. Here, V_{35} is not materialized, we simply calculate $|V_{35}|=|V_3|*|V_5|=6$. Then, $|\beta_{35}|=12-(12+6-6)=0$. As we will see later, these steps are never performed in our algorithm. The fact that c_5 does not eliminate any tuple (see $\alpha_5=0$ in Fig. 4) implies that neither do any of its super-combinations. Thus, a priori we know that $\alpha_{35}=\alpha_{235}=\ldots=0$.

Finally, we illustrate the case of a bigger size combination, for example $c_2c_3c_4$ of size 3. Eq. (E) yields $|(V_2\cup V_3\cup V_4)^{ext}|=|(V_2\cup V_4)^{ext}|+|V_3^{ext}|-|(V_2\bowtie V_3)^{ext}|-|(V_4\bowtie V_3)^{ext}|+|(V_2\bowtie V_3\bowtie V_4)^{ext}|$. All terms of the right side of the equation are available from previous iterations, except for $|(V_2 \bowtie V_3 \bowtie V_4)^{ext}|$. As before, we check the common attributes of the views and obtain $V_{234}=V_{24}\bowtie_{R\_Id,S\_Id,T\_Id}V_3$. So, $|(V_2 \cup V_3 \cup V_4)^{ext}|=6+6-2-1+0=9$ and $\beta_{234}=\alpha_{234}=12-9=3$. In the same way, we compute all the possible explanations until $c_1c_2c_3c_4c_5$.

View Materialization: when and how. To decide when and how to materialize the views for the explanations, we partition the set of the views associated with the conditions in $\mathcal{E}$. Consider the relation $\sim$ defined over these views by $V_i \sim V_j$ if the target schemas of V_i and V_j have at least one common attribute. Consider the transitive closure $\sim^*$ of $\sim$ and the induced partitioning of $\mathcal{V}_{\mathcal{E}}$ through $\sim^*$.

When this partitioning is a singleton, $V_{\mathcal{E}}$ needs to be materialized (Alg. 2, line 9). The materialization of $V_{\mathcal{E}}$ is specified by joining the views associated with the sub-conditions, which may be done in more than one way, as usual. For example, for the combination $c_2c_3c_4$, V_{234} can either be computed through $V_{23}\bowtie V_4$ or $V_{24}\bowtie V_3$ or $V_{34}\bowtie V_2$ or $V_2\bowtie V_3\bowtie V_4$... because all these views are known from previous iterations. The choice of the query used to materialize $V_{\mathcal{E}}$ is done based on a cost function. This function gives priority to materializing $V_{\mathcal{E}}$ by means of one join, which is always possible: because $V_{\mathcal{E}}$ needs to be materialized, we know that at least one view associated with a sub-combination of size $n-1$ has been materialized. In other words, priority is given to using at least one materialized view associated with one of the largest sub-combinations. For our example, it means that either $V_{23}\bowtie V_4$ or $V_{24}\bowtie V_3$ or $V_{34}\bowtie V_2$ is considered. In order to choose among the one-join queries computing $V_{\mathcal{E}}$, we favour a one-join query $V_i\bowtie V_j$ minimal w.r.t. $|V_i|+|V_j|$. For the example, and considering also Fig. 4 we find that $|V_2|+|V_{34}|=|V_4|+|V_{23}|=5$ and $|V_3|+|V_{24}|=3$. So, the query used for the materialization is $V_3\bowtie V_{24}$ (its result being empty in our example). Nevertheless, we avoid the materialization of $V_{\mathcal{E}}$ if the partitioning is a singleton (Alg. 2, line 9

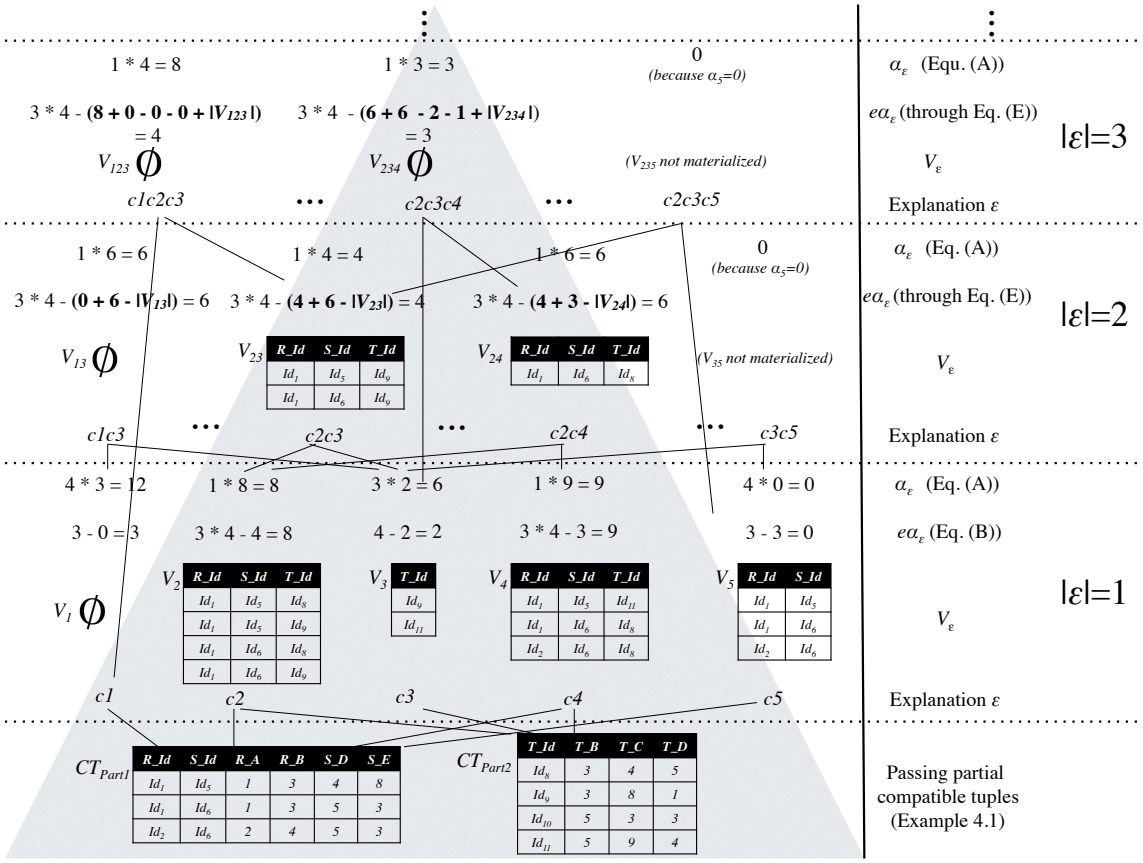


Figure 4: Running example with the different steps of *Ted++* (up to explanations of size 3) in Alg. 1 and Alg. 2

& 16), when for some sub-combination $\mathcal{E}'$ of $\mathcal{E}$ it was computed that $\alpha_{\mathcal{E}'}$=0. In that case, we know a priori that $\alpha_{\mathcal{E}}$=0 (see Ex. 4.4).

If the partitioning is not a singleton, $V_{\mathcal{E}}$ is not materialized (Alg. 2, line 14). For example, the partitioning for c_3c_5 is not a singleton and so the size $|V_{35}|$=$|V_3|\times|V_5|$=6.

Post-processing. In Alg. 2 we associated with each possible explanation $\mathcal{E}$ the number of eliminated tuples $\alpha_{\mathcal{E}}$. However, $\alpha_{\mathcal{E}}$s includes any tuple eliminated by $\mathcal{E}$, even though the same tuples may be eliminated by some super-combinations of $\mathcal{E}$ (see Ex. 4.5). This means that for some tuples, multiple explanations have been assigned. To make things even, the last step of *Ted++* (Alg. 1, line 6) is about calculating the coefficient of $\mathcal{E}$ by subtracting the coefficients of its super-combinations from $\alpha_{\mathcal{E}}$:

$$coef_{\mathcal{E}} = \alpha_{\mathcal{E}} - (\sum_{\mathcal{E} \subseteq \mathcal{E}'} coef_{\mathcal{E}'}) \quad \text{(F)}$$

EXAMPLE 4.5. *Consider known $coef_{1234}$=2 and $coef_{123}$=2. We have found in Ex. 4.4 that α_{23}=4. With Eq. (F), $coef_{23}$=4−2−2=0. In the same way $coef_2$=4−0−2−2=0. The algorithm leads to the expected Why-Not explanation polynomial already provided in Ex. 3.4.*

4.3 Complexity Analysis

In the pseudo-code provided by Alg. 1, we can see that *Ted++* divides into the phases of (i) partitioning $\mathcal{S}$, (ii) materializing a view for each partition, (iii) computing the explanations, and (iv) computing the exact coefficients. When computing the explanations, according to Alg. 2, *Ted++* iterates through $2^{|C|}$ condition combinations and for each, it decides upon view materialization (again through partitioning) before materializing it, or simply calculates $|V_{\mathcal{E}}|$ before applying equations to compute $\alpha_{\mathcal{E}}$. Overall, we consider that all mathematical computations are negligible so, the worst case complexities of steps (i) through (iv) sum up to $O(|\mathcal{S}|+|WN|)+O(|\mathcal{S}|)+O(2^{|C|}(|\mathcal{S}|+|C|))+O(2^{|C|})$. For sufficiently large queries, we can assume that $|\mathcal{S}|+|C|<<2^{|C|}$, in which case the complexity simplifies to $O(2^{|C|})$.

The complexity analysis above does not take into account the cost of actually materializing views; in its simplified form, it only considers how many views need to be materialized in the worst case. Assume that n=$max(\{|\mathcal{I}_R||R\in\mathcal{S}\})$. The materialization of any view is bound by the cost of materializing a cross product over the relations involved in the view - in the worst case $O(n^{|\mathcal{S}|})$. This yields a combined complexity of $O(2^{|C|}n^{|\mathcal{S}|})$. However, *Ted++* in the general case (more than one induced partitions), has a tighter upper bound: $O(n^{k_{\mathcal{E}1}}+n^{k_{\mathcal{E}2}}+\cdots+n^{k_{\mathcal{E}N}})$, where $k_{\mathcal{E}}$=$|part_{\mathcal{E}}|$, for all combinations $\mathcal{E}$ and $N=2^{|C|}$.

5. EXPERIMENTAL EVALUATION

We perform an experimental evaluation of *Ted++* on real and synthetic datasets. In Sec. 5.1, we compare *Ted++* to Ted [2], NedExplain [3], and Why-Not [7]. Sec. 5.2 studies the runtime of *Ted++* w.r.t. various parameters that we vary in a controlled manner. All Java implementations of the algorithms ran on MAC OS X 10.9.5 with 1.8 GHz Intel Core i5, 4GB memory, and 120GB SSD. PostgreSQL 9.3 was used as database system.

Table 2: Queries for the scenarios in Table 3

Query	Expression
Q1	$C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P$
Q2	$\sigma_{C.sector>99}[C] \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P$
Q3	$W \bowtie_{sector2} C2 \bowtie_{sector1} \sigma_{C.type=Aiding}[C]$
Q4	$P2 \bowtie_{!name,hair} \sigma_{P1.name<B}[P1]$
Q5	$L \bowtie_{movieId} \sigma_{M.year>2009}[M] \bowtie_{name} \sigma_{R.rating\geq 8}[R]$
Q6	$\sigma_{AA.party=Republican}[AA] \bowtie_{id} \sigma_{Co.Byear>1970}[Co]$
Q7	$E \bowtie_{eId} \sigma_{ES.sub=Sen.\ Com.}[ES] \bowtie_{id} \sigma_{SPO.party=Rep.}[SPO]$
Q_{s3}	$\sigma_{type=Aiding}[Q2]$
Q_{s4}	$\sigma_{witnessname>S}[Q_{s3}]$
Q_j	$C \bowtie_{sector} \sigma_{name>S}[W]$
Q_{j2}	$Q_j \bowtie_{witnessname} S$
Q_{j3}	$Q_{j2} \bowtie_{clothes} P$
Q_{j4}	$Q_{j3} \bowtie_{hair} P$
Q_c	$L1 \bowtie_{lid} L2 \bowtie_{M2.mid=L2.mid\ M2} \bowtie_{year,!mid} \sigma_{year=1980}[M1]$
Q_{tpch}	$C \bowtie_{ckey} \sigma_{odate<1998-07-21}[O] \bowtie_{okey} \sigma_{sdate>1998-07-21}[L]$

5.1 Comparative Evaluation

The comparative evaluation to Why-Not and NedExplain considers both efficiency (runtime) and effectiveness (explanation quality). When considering efficiency, we also include Ted in the comparison (Ted producing the same Why-Not explanation as *Ted++*).

For the experiments we have used data from three databases named *crime*, *imdb*, and *gov*. The *crime* database (available at http://infolab.stanford.edu/trio/) is a synthetic database about crimes and involved persons (suspects and witnesses). The *imdb* database contains real-world movie data from IMDB (http://www.imdb.com). Finally, the *gov* database contains information about US congressmen and financial activities (data from http://bioguide.congress.gov, http://usaspending.gov, and http://earmarks.omb.gov).

For each dataset, we have created a series of scenarios (crime1-gov5 in Tab. 3 - ignore remaining scenarios for now). Each scenario consists of a query further defined in Tab. 2 (Q1-Q7) and a simple Why-Not question, as Why-Not and NedExplain support only this type of Why-Not question. We have designed queries with a small set of conditions (Q6) or a larger one (Q1, Q3, Q5, Q7), containing self-joins (Q3, Q4), having empty intermediate results (Q2), as well as containing inequalities (Q2, Q4, Q5, Q6).

5.1.1 Why-Not Explanation Evaluation

Tab. 1 states that the explanations returned by Why-Not and NedExplain consist of sets of query conditions, whereas *Ted++* returns a polynomial of query conditions. For comparison purposes, we trivially map *Ted++*'s Why-Not explanation to sets of conditions, e.g., $3c_3 * c_4 + 2c_3 * c_6$ maps to $\{\{c_3, c_4\}, \{c_3, c_6\}\}$. For conciseness, we abbreviate condition sets, e.g., to c_{34}, c_{36}.

Tab. 4 summarizes the Why-Not explanations of the three algorithms. These scenarios make apparent that the explanations by NedExplain or Why-Not are incomplete, in two senses. First, they produce only a subset of the possible explanations, failing to provide alternatives that could be useful to the user when she tries to fix the query. Second, even the explanation they provide may lack parts, which can drive the user to fruitless fixing attempts. On the contrary, *Ted++* produces all the possible, complete explanations.

For the first argument, consider the scenario $gov2$. Why-Not and NedExplain return c_1 and c_3 respectively, but they both fail to indicate that both the explanations are valid, as opposed to *Ted++*. Then, consider $crime8$. NedExplain returns the join c_2 ($S \bowtie_{hair} P$) - Why-Not does not produce any explanations. *Ted++* indicates that except for this join, the selection c_3 ($\sigma_{name<'B'}[P]$) for instance is also an explanation. From a developer's perspective, selections are typically easier or more reasonable to change. So, having the complete set of explanations potentially provides the developer with useful alternatives.

Table 3: Scenarios

Scenario	Query	Why-Not question
crime1	Q1	{P.Name=Hank,C.Type=Car theft}
crime2	Q1	{P.Name=Roger,C.Type=Car theft}
crime3	Q2	{P.Name=Roger,C.Type=Car theft}
crime4	Q2	{P.Name=Hank,C.Type=Car theft}
crime5	Q2	{P.Name=Hank}
crime6	Q3	{C2.Type=kidnapping}
crime7	Q3	{W.Name=Susan,C2.Type=kidnapping}
crime8	Q4	{P2.Name=Audrey}
imdb1	Q5	{name=Avatar}
imdb2	Q5	{name=Christmas Story,L.locationId=USANew York}
gov1	Q6	{Co.firstname=Christopher}
gov2	Q6	{Co.firstname=Christopher,Co.lastname=MURPHY}
gov3	Q6	{Co.firstname=Christopher,Co.lastname=GIBSON}
gov4	Q7	{sponsorId=467}
gov5	Q7	{SPO.sponsorln=Lugar,E.camount>=1000}
$crime_s - crime_{s4}$	Q1,Q2, Q_{s3},Q_{s4}	{P.Name=Hank,C.Type=Car theft}
$crime_j - crime_{j4}$	$Q_j - Q_{j4}$,	{W.name=Jane, C.type=Car theft}
$imdb_c$	Q_{c4}	{L2.locationid=L1.locationid, M1.mid=L2.mid, L1.year>L2.year,M1.name=Duck Soup}
$imdb_{c2}$	Q_{c4}	({L2.locationid=L1.locationid, M1.mid=L2.mid, L1.year>L2.year}
$crime_{5c2}$	Q2	{P.Name=Hank, C.type=Car theft}
$crime_{5c3}$	Q2	{P.Name=Hank, C.type=Car theft, S.witness=Aphrodite}
$crime_{5c4}$	Q2	({P.Name=Hank ,C.type=Car theft, S.witness=Aphrodite, W.sector =34}
$crime_{5c5}$	Q2	{P.Name=Hank ,C.type=Car theft, S.witness=Aphrodite, W.sector =34,S.hair=green}
$imdb_{cc}$	Q_c	{M.year>M2.year}
$tpch_s$	Q_{tpch}	{L.extprice>50000,O.odate<1996-01-01}
$tpch_c$	Q_{tpch}	{L.extprice>100000, O.odate=L.cdate, C.nkey=4}

For the second argument consider $crime5$. NedExplain returns c_1 ($C \bowtie_{sector} W$). The explanation of *Ted++* does not contain the atomic explanation c_1, but there exist combinations including c_1 as a part, like c_{15}. This means that the explanation by NedExplain is incomplete; a repair attempt of c_1 alone will never yield the desired results. Similarly, $crime7$ illustrates a case, when the Why-Not algorithm produces an explanation (c_3) that misses some parts. Then, in gov3 NedExplain and Why-Not both return c_2. However, let us now assume the developer prefers to not change this condition. Keeping in mind that those algorithms' answers may change when changing the query tree, she may start trying different trees to possibly obtain a Why-Not explanation without c_2. Knowing the explanation of *Ted++* prevents her from spending any effort on this, as it shows that all explanations contain c_2 as a part.

By mapping the explanation of *Ted++* to sets of explanations, the usefulness of the coefficients of the polynomial has been neglected. For example, the Why-Not explanation polynomial of $crime8$ is $2384 * c_{23} + 20 * c_3 + 4 * c_1 + 8 * c_2$. Assume that the developer would like to recover at least 5 missing tuples, by changing as few conditions as possible. The polynomial implies to change either c_3 or c_2: they both require one condition change and provide the possibility of obtaining up to 20 and 8 missing tuples, respectively. c_1 can recover up to 4 tuples, whereas c_2c_3 require two condition changes. Clearly, the results of NedExplain or Why-Not are not informative enough for such a discussion.

5.1.2 Runtime Evaluation

***Ted++* vs. NedExplain and Why-Not.** Fig. 5 shows the runtimes in logarithmic scale for each algorithm and scenario. We observe that *Ted++* and NedExplain are comparable and that in some cases,*Ted++* is significantly faster than Why-Not.

Why-Not traces compatible tuples based on tuple lineage stored in Trio. As already stated in [3, 7], this design choice slows down Why-Not. On the contrary, both NedExplain and *Ted++* compute

Table 4: *Ted++*, Why-Not, NedExplain answers per scenario

Scenario	*Ted++*	Why-Not	NedExplain
crime1	$c_{1234}, \ldots, c_{12}, c_3, c_2, c_1$		c_1
crime2	$c_{1234}, c_{34}, c_{13}, \ldots, c_3$	c_{34}	c_{34}, c_1
crime3	$c_{12345}, \ldots, c_{145}, c_{345}, c_{35}$	c_{34},c_5	c_5, c_{34}
crime4	$c_{12345}, \ldots, c_{25}, c_{15}$	c_5	c_1, c_5
crime5	$c_{12345}, \ldots, c_{15}, c_5$	c_5	c_1
crime6	$c_{123}, c_{31}, c_{23}, c_{12}, c_3, c_2, c_1$	c_3	c_2
crime7	$c_{123}, c_{13}, c_{12}, c_1$	c_3	c_2, c_1
crime8	c_{23}, c_3, c_2, c_1		c_2
imdb1	$c_{123}, c_{13}, c_{23}, c_3$	c_3	c_3,c_2
imdb2	c_{13}		c_1, c_3
gov1	$c_{123}, c_{13}, c_{23}, c_{12}, c_3, c_2, c_1$	c_3	c_2, c_3
gov2	c_{13}, c_3, c_1	c_1	c_3
gov3	c_{123}, c_{23}, c_2	c_2	c_2
gov4	c_{123}, c_{23}, c_2	c_3	c_3, c_2
gov5	$c_{124}, c_{14}, c_{24}, c_{12}, c_4, c_2, c_1$	c_1	c_1

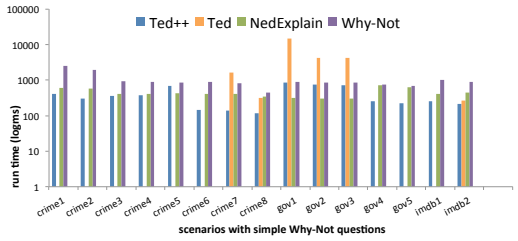


Figure 5: Runtime for *Ted++*, Ted, NedExplain, and Why-Not

compatible data more efficiently. We claim that a better implementation choice for tuple tracing in Why-Not would yield a runtime comparable to NedExplain, a claim backed up by their comparable runtime complexities. Another problem of NedExplain and Why-Not lies in the choice to trace compatible data w.r.t. tuples from the input relations but not necessarily compatible ones.

Let us see what happens when *Ted++* is slower than - but still comparable to - NedExplain, for example in $gov1$-$gov3$. In NedExplain all compatible tuples are pruned out by conditions very close to the leaf level of the query tree, so the bottom-up traversal of the tree can stop very early. *Ted++* always "checks" all conditions so cannot benefit from such an early termination. However, this runtime improvement of NedExplain often comes at the price of incomplete explanations (e.g., $gov1$).

***Ted++* vs. Ted.** Fig. 5 reports runtimes for Ted on 6 out of 15 scenarios as for the others, Ted runs out of time. To examine this behavior, we compare the time distribution in Ted and *Ted++*(Fig. 6). The algorithms are divided in four common phases. Note that in scenarios $crime7$, $gov1 - gov3$ the diagram bars for Ted are not totally displayed as the execution time is much higher compared to the other scenarios and to the runtime of *Ted++* (the runtime of the coefficientEstimation phase is the label on the respective bars) .

As said in Sec. 4, Ted's main issue is its dependence on the number of compatible tuples. This is experimentally observed in Fig. 6: with the growth of the set of compatible tuples, the time dedicated to coefficientEstimation also grows (the scenarios are reported in an ascending order of number of compatible tuples). The number of compatible tuples affects *Ted++* too, but not as much. This can be seen in $crime8$ and $crime7$, or $gov3$ and $gov1$; while the number of tuples grows, *Ted++*'s runtime remains roughly steady.

5.2 Ted++ Analysis

We now study *Ted++*'s behavior w.r.t. the following parameters: (i) the type (simple or complex) of the input query Q and the number of Q's conditions, (ii) the type of the Why-Not question (simple

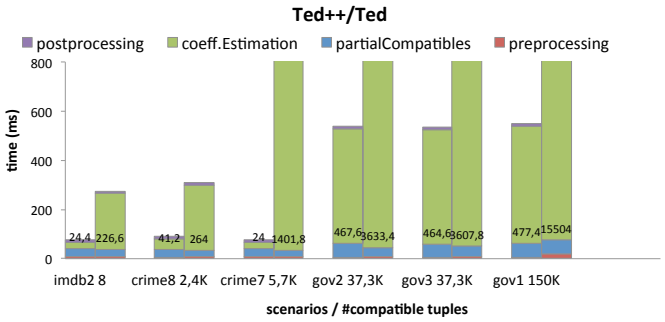
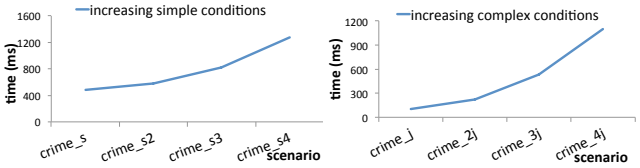


Figure 6: *Ted++* and Ted runtime distribution



(a) simple conditions (b) complex conditions

Figure 7: *Ted++* runtime w.r.t. number of conditions in Q

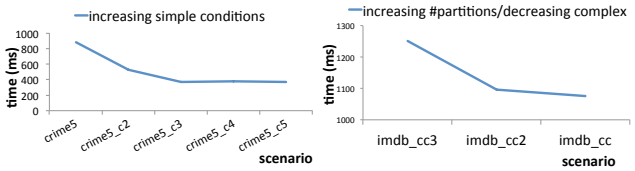
or complex) and the number and selectivity of conditions the Why-Not question involves, and (iii) the size of the database instance $\mathcal{I}$. Note that (ii) and (iii) are tightly connected with the number of compatible tuples, which is one of the main parameters influencing the performance. Another important factor is the selectivity of the query conditions over the compatible data.

For the parameter variations (i) and (ii), we use again the *crime*, *imdb*, and *gov* databases. To adjust the database instance size for case (iii), we use data produced by the TPC-H benchmark data generator (http://www.tpc.org/tpch/). We have generated instances of 1GB and 10GB and further produced smaller data sets of 10MB and 100MB to obtain a series of datasets whose size differs by a factor of 10. In this paper, we report results for the original query Q3 of the TPC-H set of queries. It includes two complex and three simple conditions, two of which are inequality conditions. Since the original TPC-H query Q3 is an aggregation query, we have changed the projection condition. The queries used in this section are Q_s-Q_{tpch} (Tab. 2) and the scenarios are $crime_s$-$tpch_c$ (Tab. 3).

Adjusting the query. Given a fixed database instance and Why-Not question, we start from query Q1 and gradually add simple conditions, yielding the series of queries Q1, Q2, Q_{s3}, Q_{s4}. The evolution of *Ted++* runtime for these queries is shown in Fig. 7 (a). Similarly, starting from query Q_j, we introduce step by step complex conditions, yielding Q_j-Q_{j4}. Corresponding runtime results are reported in Fig. 7 (b).

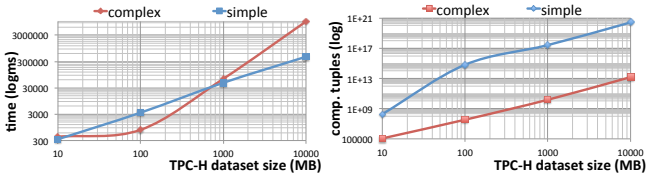
As expected, in both cases, increasing the number of query conditions (either complex or simple) results in increasing runtime. The incline of the curve depends on the selectivity of the introduced condition; the less selective the condition the steeper the line becomes. This is easy to explain, as the view for the explanations involving a low selective condition contains more tuples (=passing partial tuples). This, leaves space for further optimization by dynamically deciding on passing vs eliminated tuples materialization.

Adjusting the Why-Not question. The scenarios considered for Fig. 8 (a) have as starting point the simple Why-Not question of $crime5$ (see Tab. 3). Keeping the same input instance and query, we add attibute-constant comparisons (i.e., simple conditions) to WN, resulting in fewer tuples in each step. As expected, the more conditions (the less tuples) the faster the Why-Not explanation is returned, until we reach a certain point (here from $crime5_{c3}$ on).



(a) simple conditions (b) complex conditions

Figure 8: *Ted++* runtime w.r.t. number of conditions in WN



(a) runtime (b) # compatible tuples

Figure 9: *Ted++* behavior for varying database size

From this point, the runtime is dominated by the time to communicate with the database that is constant over all scenarios.

In Fig. 8 (b) we examine complex Why-Not questions. As we add complex conditions in a Why-Not question, the number of generated partitions (potentially) drops as more relations are included in a same partition. To study the impact of the induced number of partitions in isolation, we *keep the number of the compatible tuples constant* in our series of complex scenarios ($imdb_{cc}$-$imdb_{cc3}$). The number of partitions entailed by $imdb_{cc}$, $imdb_{cc2}$, and $imdb_{cc3}$ are 3, 2, and 1, respectively. The results of Fig. 8 (b) confirm our theoretical complexity discussion, i.e., as the number of partitions decreases, the time needed to produce the Why-Not explanation increases.

Increasing size of input instance. Now we increase the database size for scenarios with one simple or one complex Why-Not question WN, over the same query Q_{tpch}. The simple WN includes two inequality conditions, in order to be able to compute a reasonable number of compatible tuples. The complex WN contains one complex condition, one inequality simple condition and one equality simple condition. It thus represents an average complex Why-Not question, creating two partitions over three relations.

Fig. 9 (a) shows the runtimes for both scenarios. The increasing runtime is tightly coupled to the fact that the number of computed tuples is augmenting proportionally to the database size, as shown in Fig. 9 (b). We observe that for small datasets (<500MB) in the complex scenario *Ted++*'s performance decreases with a low rate, whereas the rate is higher for larger datasets. For the simple scenario, runtime deteriorates in a steady pace. This behavior is aligned with the theoretical study; when the number of partitions is decreasing the complexity rises.

In summary, our experiments have shown that *Ted++* generates a more useful and complete Why-Not explanation than the state of the art. Moreover, *Ted++* is competitive in terms of runtime. The dedicated experimental evaluation on *Ted++* verifies that it can be used in a large variety of scenarios with different parameters. Finally, the fact that the experiments were conducted on an ordinary laptop supports *Ted++*'s feasibility.

6. CONCLUSION AND OUTLOOK

This paper provides a framework for Why-Not explanations based on polynomials, which enables to consider relational databases under set, bag and probabilistic semantics in a unified way. To efficiently compute the Why-Not explanation polynomial under set semantics we have designed a new algorithm *Ted++*, whose main feature is to completely avoid enumerating and iterating over the set of compatible tuples, thus significantly reducing both space and time consumption. Our experimental evaluation showed that *Ted++* is at least as efficient as existing algorithms while providing useful insights in its Why-Not explanation for a developer. Also, we show that *Ted++* scales well w.r.t various parameters, making it a practical solution.

Why-Not explanation polynomials are easy to extend for unions of conjunctive queries, whereas an extension for aggregation queries is subject to future work. Currently, we have been working on exploiting the Why-Not explanation polynomial to efficiently rewrite a query in order to include the missing answers in its result set. As there are many rewriting possibilities, we plan to select the most promising ones based on a cost function, built with the polynomial. For instance, we may rank higher rewritings with minimum condition changes (i.e., small combinations), minimum side-effects (i.e., small coefficients), etc.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *Workshop on Theory and Practice of Provenance (TAPP)*, 2014.

[3] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based why-not provenance with NedExplain. In *International Conference on Extending Database Technology (EDBT)*, 2014.

[4] N. Bidoit, M. Herschel, and K. Tzompanaki. EFQ: Why-not answer polynomials in action. *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[5] D. Calvanese, M. Ortiz, M. Simkus, and G. Stefanoni. Reasoning about explanations for negative query answers in DL-Lite. *Journal on Artificial Intelligence Research (JAIR)*, 2013.

[6] B. t. Cate, C. Civili, E. Sherkhonov, and W.-C. Tan. High-level why-not explanations using ontologies. In *Principles of Database Systems (PODS)*, 2015.

[7] A. Chapman and H. V. Jagadish. Why not? In *International Conference on the Management of Data (SIGMOD)*, 2009.

[8] L. Chen, X. Lin, C. S. Jensen, and J. Xu. Answering why-not questions on spatial keyword top-k queries. In *International Conference on Data Engineering (ICDE)*, 2015.

[9] Y. Gao, Q. Liu, G. Chen, B. Zheng, and L. Zhou. Answering why-not questions on reverse top-k queries. *PVLDB*, 8(7):738–749, 2015.

[10] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Principles of Database Systems (PODS)*, 2007.

[11] M. Hall. *Combinatorial theory*, volume 71. John Wiley & Sons, 1998.

[12] Z. He and E. Lo. Answering why-not questions on top-k queries. In *International Conference on Data Engineering (ICDE)*, 2012.

[13] M. Herschel. Wondering why data are missing from query results? ask Conseil Why-Not. In *International Conference on Information and Knowledge Management (CIKM)*, 2013.

[14] M. Herschel. A hybrid approach to answering why-not questions on relational query results. *Journal on Data Information and Quality*, 5(3):10:1–10:29, 2015.

[15] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[16] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[17] M. S. Islam, C. Liu, and R. Zhou. Flexiq: A flexible interactive querying framework by exploiting the skyline operator. *Journal of Systems and Software*, 97:97–117, 2014.

[18] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *International Conference on Data Engineering (ICDE)*, 2013.

[19] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment (PVLDB)*, 2011.

[20] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *International Conference on the Management of Data (SIGMOD)*, 2010.

[21] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query reverse engineering. *The VLDB Journal*, 23(5):721–746, 2014.

[22] R. Vaught. *Set Theory An Introduction*. Birkhaeuser, 2001.